第3章 CPUコア搭載FPGAの効率的活用法

―― Virtex-4 FXのコプロセッサ機能を使いこなす

井倉将実

本稿では，CPUコアのハード・マクロを搭載するFPGAの効果的な活用法を解説する．CPUとユーザ回路がそれぞれ別のLSIになる場合と，一つのLSIに集積できる場合の違いを説明する．米国Xilinx社のVirtex-4 FXが持つハードウェア・アクセラレータ機能の利用法とアプリケーション事例を紹介する．（編集部）

従来からよく，ソフトウェアとハードウェアの境界線について設計時に取りざたされます．CPUコア内蔵のFPGAは，最適な設計を1チップで実現できるLSIと言えます．例えば回路全体を制御するステート・マシンの役割をCPUに割り当てたり，ハードウェアでは複雑すぎて設計に時間がかかるところをCPUによってエミュレーションするという使いかたが考えられます．設計資産を活用しやすくなるなど，さまざまな利点があります．

従来は，汎用CPUでA機能，FPGAでB機能，ASSPでC機能といったぐあいに割り当てて，一つのシステムを設計していました．これが1チップでできると，プリント基板面積の削減や低コスト化につながります．さらに回路構成が外部から見えにくくなるため，機密保持の点でもメリットがあります．

本稿では，CPUコアのハード・マクロを内蔵するFPGAを効果的に活用する方法を解説します．ユーザ回路とCPUの間の接続についての考えかたや，米国Xilinx社のVirtex-4 FXファミリが持つハードウェア・アクセラレータ（コプロセッサ）機能APU（Auxiliary Processing Unit）を用いる方法を説明します．

3

1．FPGA＋CPUコアによる性能の向上

FPGAに内蔵されるCPUコアであっても，設計の考えかたは汎用CPUを使う場合と変わりません．CPUがプリント基板上でつながっているのか，シリコン上でつながっているのかの違いだけです．ただし，シリコン上のほうが配線に制約が少ないので，バス構造やリセット/割り込み回路などの自由度が高くなることが多いようです．

DCR ：Device Control Register
FCB ：Fabric Coprocessor Bus
OCM：On-Chip Memory
PLB ：Processor Local Bus

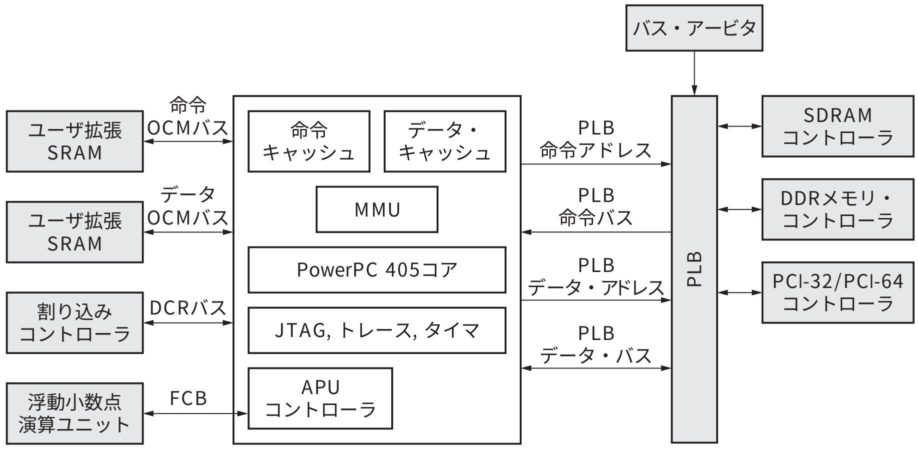


図1 Virtex-4 FXに内蔵されるPowerPCコアのブロック図
命令バスとデータ・バスが独立している．灰色の部分はユーザ回路．

● **FPGAに組み込まれるCPUコアには制約が少ない**

例えば，組み込み向けCPUとして販売されている米国Applied Micro Circuits社（AMCC）の「PowerPC 405GPr」のバス構成と，Xilinx社のFPGA「Virtex-4 FX」が内蔵するPowerPC 405コア（以下，単に「PowerPCコア」と呼ぶ）を比較してみます．**図1**に示すPowerPCコアのバスは，**図2**に示すPowerPC 405のバスと比較して，自由度が高いことがわかります．

PowerPC 405GPrでは，命令バスとデータ・バスが一つにまとめられているため，例えば外部DMAコントローラのアクセス中は，命令アクセスが一時的に待たされることになります．これに対してPowerPCコアでは，例えば命令バスにフラッシュ・メモリを，データ・バスにSRAMとDMAコントローラをつなげることができます．PowerPCコアは，命令を読み出すために命令バスに接続されたフラッシュ・メモリにアクセスします（フェッチ・サイクル）．命令をフェッチしている限りにおいては，データ・バス上のDMAコントローラはSRAMメモリ空間を自由にアクセセ

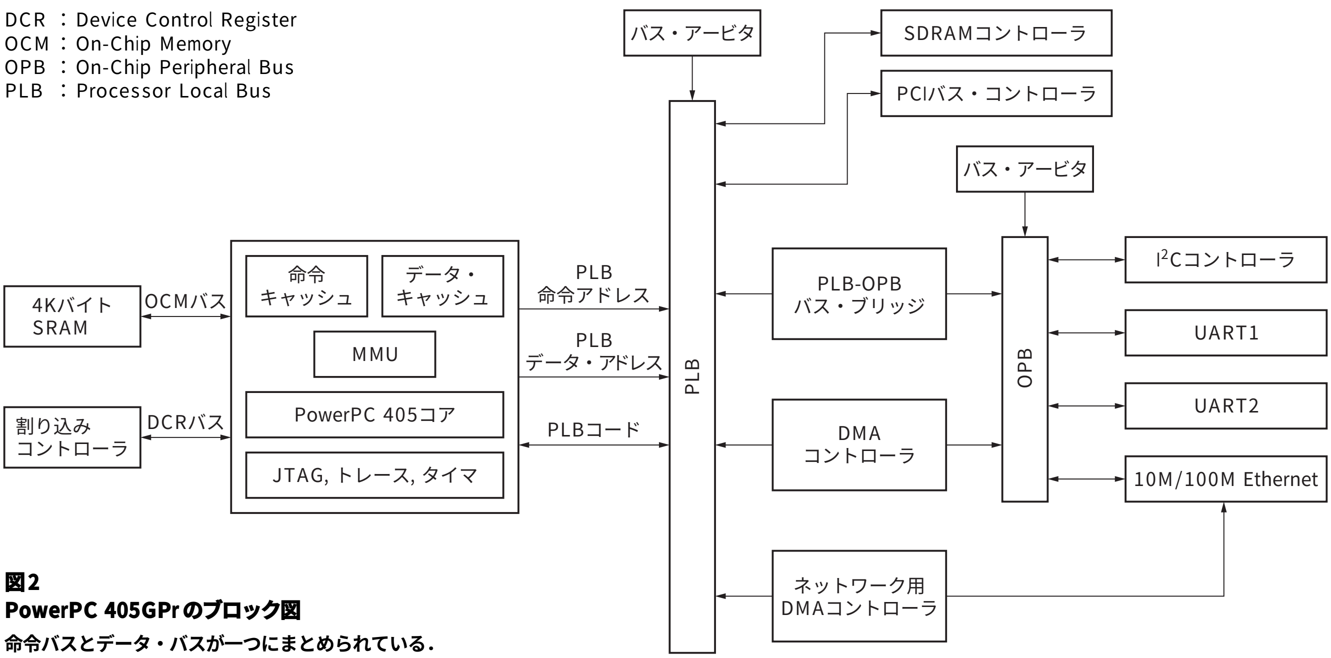


図2
PowerPC 405GPrのブロック図
命令バスとデータ・バスが一つにまとめられている．

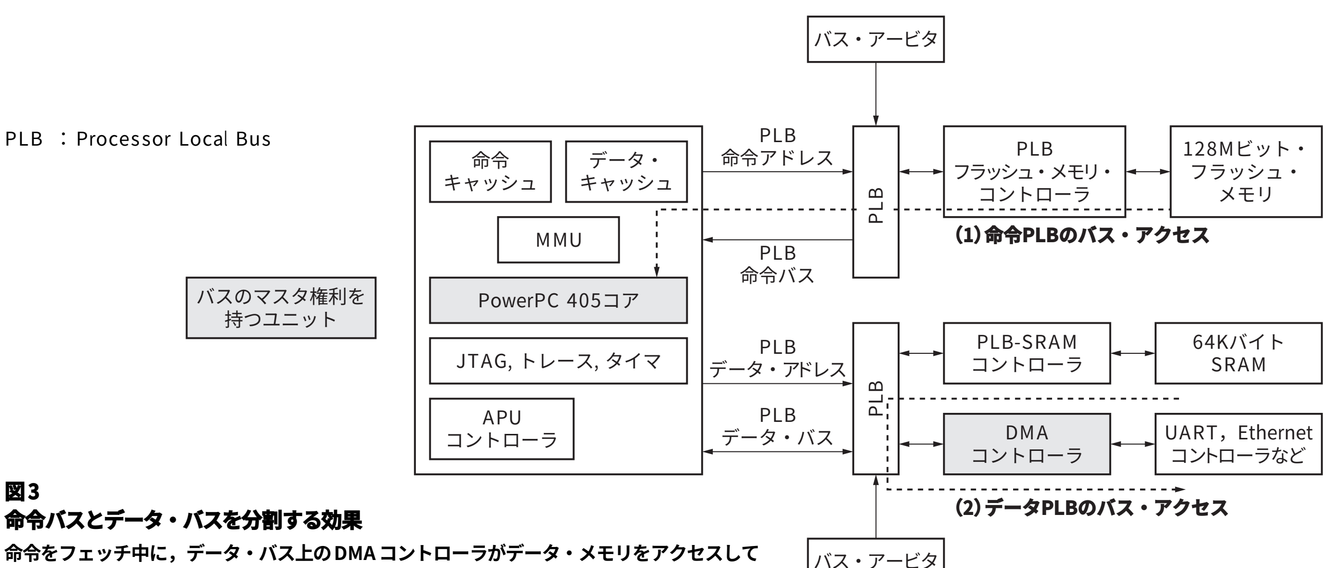


図3
命令バスとデータ・バスを分割する効果
命令をフェッチ中に，データ・バス上のDMAコントローラがデータ・メモリをアクセスしても競合しない．

スできます．つまり，PowerPCコアとDMAコントローラは同一時間に並列して動くことができます（**図3**）．

もう少し具体的に，例えば行列演算コプロセッサを考えてみましょう．CPUからの処理開始要求を受けると，データ・バス上のメモリからソース値を読み出し，演算後に結果をメモリに書き戻し，CPUに処理終了通知を割り込みという形で返すものとします．外部メモリからソース値を読み出したり処理結果を書き戻す際に，DMAコントローラを使います．PowerPC 405GPrのように命令バスとデータ・バスが共通している一般的なCPUでは，メモリ・アクセスの際に命令フェッチが強制的に停止（デッドタイムが発生）します．しかし，PowerPCコアのように命令バスとデータ・バスが独立していれば，このようなデッドタイムは発生しません．

● バス接続の問題

LSIの中でCPUコアとユーザ回路を接続する場合，ボード上の回路と同じように，バスを使うのが一般的です．標準化されたバス向けには，機能検証済みのIPコアが数多く提供されており，あたかもボード上でCPUと周辺LSIを組み合わせるかのようにLSIを設計できます．これは設計期間の短縮に貢献します．

バスに接続したユーザ回路とCPUのインターフェースは，メモリ空間に割り当てたレジスタを読み書きすることで行います．ソフトウェアでも扱いやすい構成です．しかし，ユーザ回路が高度なOSによって管理される場合には事情が異なります．メモリ空間をソフトウェアがアクセスするとき，論理アドレスから物理アドレスへの変換が行われるためです．このため，デバイス・ドライバを使うことになり

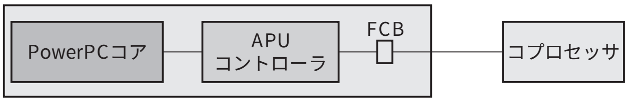


図4　APUの概要
CPUの機能を補完するハードウェア・アクセラレータを接続するインターフェースである．

3

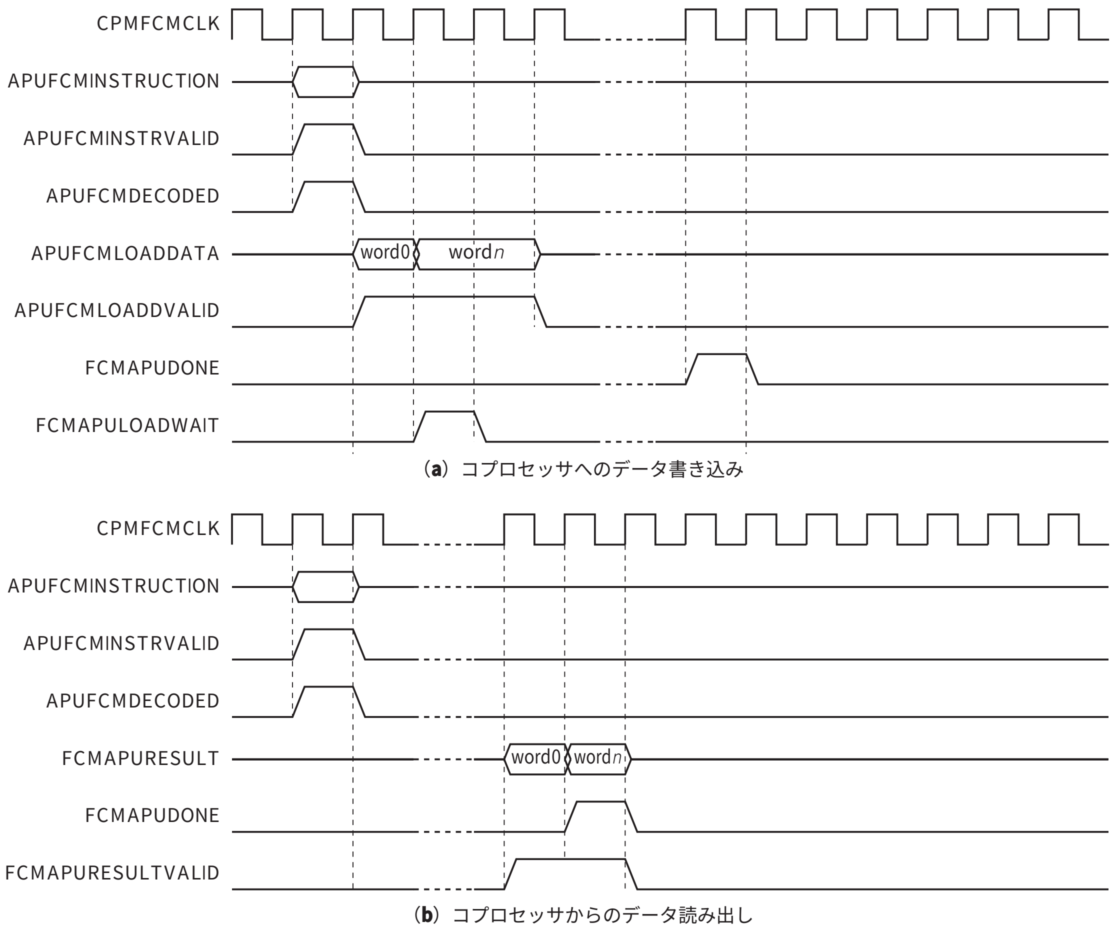


図5
APU-コプロセッサ・インターフェース
一般的な32ビット・データ・バスとあまり変わらない．

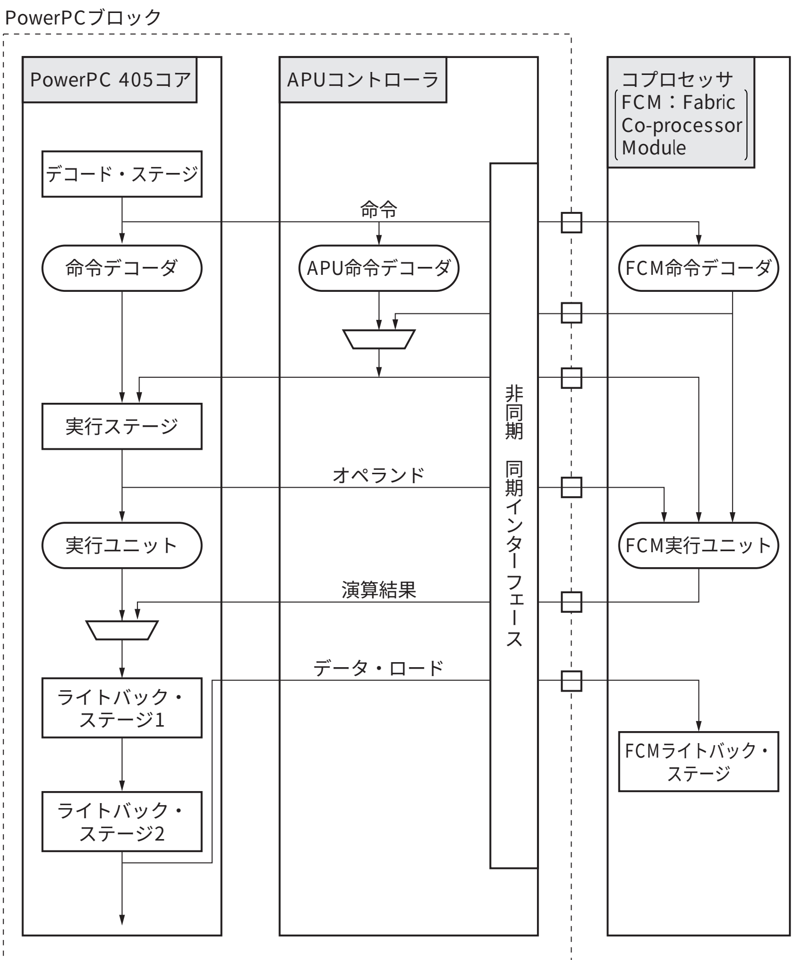


図6　APUの動作
APU命令によりコプロセッサをアクセスする．

表1　代表的なAPU命令

機　能	命　令	サイズ
インデックス加算付きロード	lbfcmux	1バイト
	lhfcmux	2バイト
	lwfcmux	4バイト
	ldfcmux	8バイト
	lqfcmux	16バイト
ロード	lbfcmx	1バイト
	lhfcmx	2バイト
	lwfcmx	4バイト
	ldfcmx	8バイト
	lqfcmx	16バイト
インデックス加算付きストア	stbfcmux	1バイト
	sthfcmux	2バイト
	stwfcmux	4バイト
	stdfcmux	8バイト
	stqfcmux	16バイト
ストア	stbfcmx	1バイト
	sthfcmx	2バイト
	stwfcmx	4バイト
	stdfcmx	8バイト
	stqfcmx	16バイト

注：1ワード＝4バイト

ますが，性能面ではオーバヘッドとなってしまいます．

組み込みシステムの設計では，CPUとソフトウェアによる構成が要求仕様にマッチしていれば，C/C++などでコードを書いて終わりです．これでは要求仕様を満たせないときに，専用ハードウェアによる処理を考えます．ところが高度なOSを使うと，デバイス・ドライバによるアドレス変換のオーバヘッドで性能が上がらないという問題が出てくるのです．

2．APUによるコプロセッサ機能の実現法

Xilinx社のFPGA「Virtex-4 FX」が持つ特徴的な機能の一つに，コプロセッサ機能を実現するAPUがあります．筆者は本FPGAが搭載された評価キットを使い，APUを評価してみました．

● 従来の接続方法と何が違うのか

Virtex-4 FXには，APUとFCB（Fablic Control Bus）という，CPUの機能を補完するハードウェア・アクセラレータ（本稿では「コプロセッサ」と呼ぶ）を接続するインターフェースが実装されています．FCBにはユーザが設計したコプロセッサを接続することができます（**図4**）．

APU-コプロセッサ・インターフェースを**図5**に示します．これだけを見ると単なる32ビット・データ・バスで，今までのユーザ回路接続法と何が違うのか？という疑問がわくかもしれません．もっとも大きな違いは，Virtex-4 FX内蔵のCPUコアに，FCBにつながったコプロセッサをアクセスするための専用命令が用意されていることです．C言語などからこの命令を使うとAPUが動作し，コプロセッサがアクセスされます（**図6**）．

● 3種類のAPU命令

代表的なAPU命令を**表1**に示します．このAPU命令には3種類の命令実行形態があります．

1）Autonomous

APU命令を実行したあと，FCBに接続されたユーザ回路の処理が完了していなくても，CPUは次の命令を実行できます．

例えば，APU命令を発行してユーザ回路に値をセットしたあと，CPUは引き続き次の命令を実行します．APU命令でセットされた値を使ってユーザ回路は処理を行います．この瞬間だけをとらえてみると，ユーザ回路とCPUは完全に並列処理状態になり，ハードウェアとソフトウェアがいちばん効率的に協調動作していることになります．

具体的な使用例としては，ハードウェア側の処理が数クロック～数十クロックにわたって必要であるため，その間にソフトウェアで別の処理を実行したり，直前に発行したAPU命令によるハードウェア処理結果をソフトウェアが参照することがないときに，Autonomous形態（自立的形態）の命令を発行して並列処理化を図ることが考えられます．

2）Non-Autonomous/Blocking

APU命令を実行したあと，FCBに接続されたユーザ回路が実行完了するまで，CPUは次の命令実行に移りません．

例えば，**リスト1**でｇという値は直前のｆの値が決まらない限り実行することができません．**リスト1**を正しく動作させるためには，「完了を待つAPU命令」を使って処理します．APU命令の発行後は，FCBのFCMAPUDONE（処理完了通知信号，**図5**参照）を監視し，ユーザ回路が完了したことを検出したあと，次のｇの計算を実行します．

また，CPUがユーザ回路の実行完了を待ち続ける間に割り込みが来たとしても，応答しません．割り込み処理の開始は，ユーザ回路の実行完了後に行われることになります．

3）Non-Autonomous/Non-Blocking

Non-Autonomous/Blockingと同じで，ユーザ回路からの完了通知を待って次の命令を実行します．ただし，ユーザ回路からの実行完了を待ち続ける間に割り込みが来たときの動作が異なります．

APU命令の実行中にCPUに対して外部割り込み要求が発行された際，いったんAPU命令実行を中断して割り込み処理に応答します．そして割り込み処理が完了した後，中断したAPU命令を引き続き継続実行し，ユーザ回路からの実行完了通知を待ちます．

リスト1　命令の実行形態

```
int main(void)
{
    int                 a,b,c,d,e,f ;

    c = a * b ;         // 演算の開始

    f = d * e ;         // 直前のデータを参照しない
                        //=> Autonomous形態

    g = c + f ;         // 直前のデータ(f値)を参照する
                        //=> NonAutonomous形態

    return(g) ;         // 直前のデータ値を出力する

}
```

● APU命令の使い分け

どの命令実行形態を使うかは，ユーザ回路の動作とソフトウェア・コードの書きかたに依存します．

例えば，一般的なコプロセッサ命令というと浮動小数点演算命令の実行が考えられますが，このような場合にはNon-Autonomous/Blockingを使うことになりそうです．演算結果を次の命令で再利用する必要があるためです．

メモリ間コピーとデータ処理を並列して行うような数値演算アプリケーションの場合は，Autonomous実行形式を選択することで並列化が可能であり，最大の性能を得ることができそうです．ビット・リバース処理，バタフライ演算，FFT（fast fourier transform），FIR（finite impulse response）/IIR（infinit impulse response）フィルタなど，ディジタル信号処理では頻繁に見かけます．

なお，APU命令のほとんどは，データPLBやデータOCM，データ・キャッシュ内とFCBの間のやり取りになります（**図7**）．つまり，あらかじめメモリ上にデータを用意しておいて，参照アドレスをAPU命令に渡せば，内部のシーケンサが勝手にバス間で転送してくれます．書き出しのデータ空間をキャッシャブル空間にしておけば，次回に高速に使いまわすこともできます．

● 独自命令の追加

APU命令では，ユーザ独自のコードを構築することもできます．

既存のAPU命令はあくまでも単一レジスタ空間へのロード/ストアしかサポートしておらず，ユーザ回路が複数の空間を持っているような場合には，完全な対応ができません．これに対してユーザ独自のAPU命令は，対応する

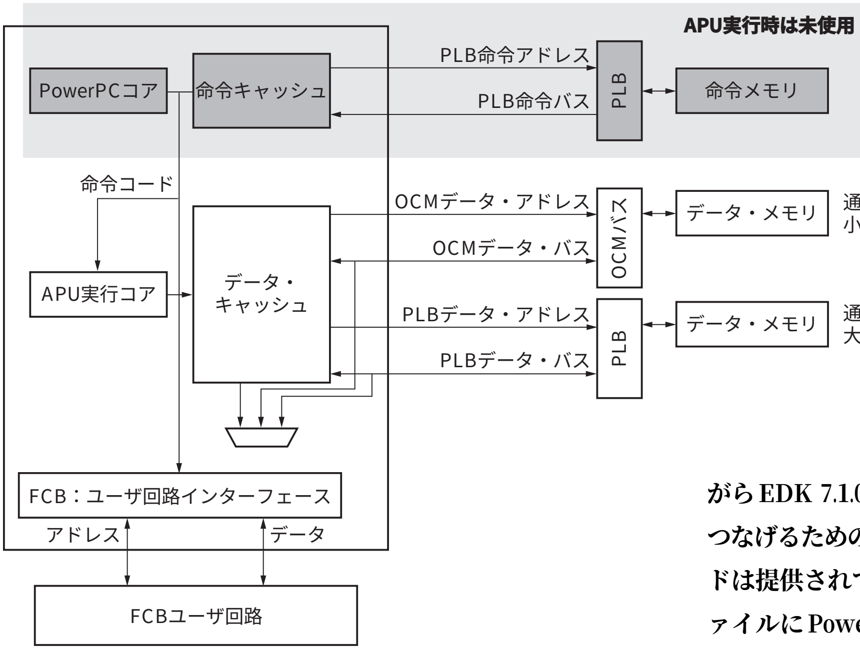


図7
APU命令実行時のデータの流れ
APU命令のほとんどは，データPLBやデータOCM，データ・キャッシュ内とFCBの間のやり取りになる．

コンパイラの改良が必要となりますが，最大で32個の32ビット長アドレス空間のいずれかを指定してアクセスを行うことができます．

3．APUを活用した半透明合成回路の設計

今回は，**図8**にあるような二つの画像を一つの画像にするミキシング機能をハードウェアで実現してAPU機能の有用性を評価しました．

● 開発環境

半透明合成回路の設計に当たって使用した開発環境について簡単にまとめます．

1）APUリファレンス・マニュアル

APUを使う設計の参考になるドキュメントは，本稿執筆時点でXilinx社が提供するアプリケーション・ノート(1)しかありませんでした．そこで，このアプリケーション・ノートを頼りに，APUバスの使用方法や命令実行状況などをシミュレータと実機で検証し，半透明合成回路を組み込みました．

2）開発ツール

一般的に開発キット（EDK）上でユーザ回路を接続する場合には，Core Imprement Wizardというツールを使って比較的簡単に実装することができます．しかし，残念ながらEDK 7.1.03の環境ではPowerPCコアとユーザ回路をつなげるためのFCBインターフェースを包括したウィザードは提供されていませんでした．そのため，直接MHSファイルにPowerPC・ユーザ回路間のポートを記述してユーザ回路を接続しました．PBDファイルを使ったグラフィカルなブロック・ダイアグラム表示ツールを起動すると，EDK全体が強制終了してしまいます．

アプリケーション・ノート(1)には，FCB経由で最大4ワードまでのデータ・ストアとデータ・ロードをサポートするVerilog HDLコードが用意されています．EDKにユーザ・ライブラリとして登録するためには，これと対にMPOファイル，PAOファイルの2種類を開いて，名称を変更します．

● 半透明合成回路について

半透明合成回路FCM_ABLENDは，FCB経由で送られた8ワード分のデータから，4画素分の半透明合成結果を算出し，その結果をCPUに返すというものです．

FCB_ABLEND回路内には三つのレジスタ空間と，転送元画素データ保持レジスタsrcPIX_REG［3：0］と転送先画素データ保持レジスタdstPIX_REG［3：0］の2組の4ワード・レジスタがあります．FCM空間2がアクセスされたことを認識すると，4ワード分の転送元画素データを転送元画素データ保持レジスタに保持します．FCM空間3がアクセスされた場合には，転送先画素データが送り込まれることになり，先に保持した転送元画素データとまさにいま送り込まれる転送先画素データの二つの画素から半透明合成処理を行って転送先画素データ・レジスタに順番に保持します．最後にFCM空間4がアクセスされると，転送先画素データ・レジスタ内部に保持された，半透明合成処理

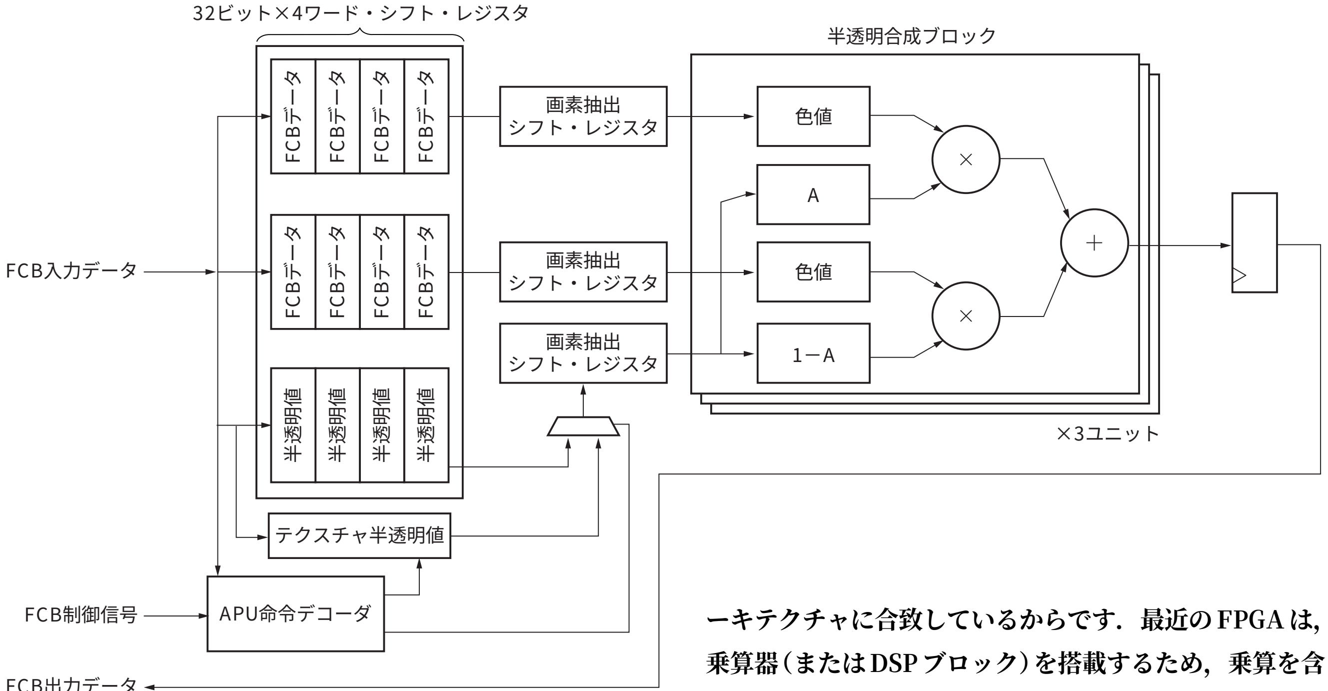


図8　半透明合成回路のブロック図
二つの画像を一つの画像にするミキシング回路である．

結果をCPUに返します．

半透明合成処理に使用される半透明の度合い（α値と呼ぶ）は，転送元画素データの最上位バイトに格納されています．

半透明合成回路の演算は，おもに以下の2種類が使用されます．

- ソフトウェア向きの公式

dstPIX［R/G/B］＝srcPIX［α］・（srcPIX［R/G/B］－dstPIX［R/G/B］）＋dstPIX［R/G/B］

dstPIX［α］＝srcPIX［α］

- ハードウェア向きの公式

dstPIX［R/G/B］＝（1－srcPIX［α］）・srcPIX［R/G/B］＋ srcPIX［α］・dstPIX［R/G/B］

dstPIX［α］＝srcPIX［α］

ソフトウェア向きのほうは，掛け算を1回しか使用していません．一般に，組み込みCPUでは加算/減算に比べて乗算は遅いためです．これに対してハードウェア向きの公式は，2回の掛け算を使用しています．これはFPGAのアーキテクチャに合致しているからです．最近のFPGAは，乗算器（またはDSPブロック）を搭載するため，乗算を含む演算とも相性がよくなっています．

半透明合成は，まず内部のステート・マシンの指示により32ビットの転送元/転送先画素データからα値とR/G/Bの8ビット単位の要素を切り出し，積和演算器に送り込まれてから結果を転送先画素データ保持レジスタに書き込みます．このとき，半透明合成で使用したα値は転送先画素の新しいα値としていっしょにレジスタに書き込まれます．保持レジスタ内のデータはCPUからストア命令によってCPUに送り込まれます．

● ソフトウェアの実装について

CPUからの送り出しは8ワード，読み出しは4ワードとなるので，4ワード・ロード命令（`lqfcmx`），ならびに4ワード・ストア命令（`stdfcmx`）を使用します．ともに，FCMに接続されたユーザ回路に対して4ワード単位でアドレス空間を指定できるので，FCMフィールドに書き込み/読み出し先の空間を指定しています．

先にも書いたことですが，FCM_ABLEND回路を操作するAPU命令はかならず4ワードのロード/ストア命令でなければなりません．バイト，ハーフ・ワード，ワードとダブル・ワードのデータ・サイズで転送を行うと正常に動作しない仕様となっています．

半透明合成すべき画像の大きさは，640×480画素（VGA）です．このサイズの絵が転送元と転送先で二つあり，全画面の半透明処理を行うためのソフトウェアを作ることにな

リスト2　半透明処理プログラム

```
#include “xbasic_types.h”
#include “xcache_l.h”
#include “xparameters.h”
#include “xpseudo_asm.h”
#include “xutil.h”

//#include “APUinst.h”

// Data structures

volatile Xint32 __attribute__ ((aligned (32)))
                                          src2[640*480] ;
volatile Xint32 __attribute__ ((aligned (32)))
                                          dst2[640*480] ;

int main(void)
{
    Xint32 x,y ;
    Xint32 src_ptr, dst_ptr, src_offset, dst_offset ;
    Xint32 srcPIX, dstPIX ;
    Xint32 srcA, srcR, srcG, srcB ;
    Xint32       dstR, dstG, dstB ;
    Xint32 resA, resR, resG, resB ;

  // initialize caches
    XCache_EnableDCache(0x80000001);
    XCache_EnableICache(0x80000001);

  // initialize APU
//   mtmsr(XREG_MSR_APU_AVAILABLE);
                                   // コプロセッサ命令の有効化

    src_offset = 0x04000000 ;   // 転送元VRAMアドレスを指定
    dst_offset = 0x04800000 ;   // 転送先VRAMアドレスを指定

    for ( y=0; y<480; y++) {
        src_ptr = y*640 + src_offset ;
        dst_ptr = y*640 + dst_offset ;

        for ( x=0; x<640; x++) {
            srcPIX = src2[src_ptr + x] ;
            dstPIX = dst2[src_ptr + x] ;

            srcA = ( srcPIX & 0xFF000000 ) >> 24 ;
            srcR = ( srcPIX & 0x00FF0000 ) >> 16 ;
            srcG = ( srcPIX & 0x0000FF00 ) >> 8 ;
            srcB = ( srcPIX & 0x000000FF ) >> 0 ;

            dstR = ( dstPIX & 0x00FF0000 ) >> 16 ;
            dstG = ( dstPIX & 0x0000FF00 ) >> 8 ;
            dstB = ( dstPIX & 0x000000FF ) >> 0 ;

            resA = srcA ;
            resR = ( ( ( 257-srcA ) * srcR ) +
                         ( ( srcA+1 ) * dstR ) ) / 256 ;
            resG = ( ( ( 257-srcA ) * srcG ) +
                         ( ( srcA+1 ) * dstG ) ) / 256 ;
            resB = ( ( ( 257-srcA ) * srcB ) +
                         ( ( srcA+1 ) * dstB ) ) / 256 ;

            dst2[src_ptr + x] = ( ( resA << 24 ) |
                                  ( resR << 16 ) |
                                  ( resG << 8 )  |
                                    resB
                                 ) ;
        }
    }

    return 0;

}
```

（a）汎用的なC言語記述

ります．

リスト2に，VGAサイズ全画面半透明処理のソース・コードを示します．2画面分の映像は4画素ごとにAPU映像処理エンジンに送られますから，640×480の1/4回を1画面として，これの3回分をデータ入出力することになります．ソフトウェアのほうは1画素ごとに処理するので，640×480×3回分の処理を行います．

● 性能の評価

設計結果から性能を比較してみました．

- CPU命令のみ：64画素処理当たり3488クロック
- CPU命令＋APU命令：64画素処理当たり456クロック

CPU命令＋APU命令による処理のほうが7.56倍ほど早いことがわかりました．いずれも命令/データ・キャッシュを有効にしており，すべてのデータはSDRAM上に格納されているので，命令フェッチの条件は同じです．APU命令を使うと，まず4ワードごとにデータ・キャッシュの内容を自動的に送り出してくれること，FCM_ABLEND回路側にはRGBごとに乗算器を用意しているのでソフトウェアでは必要なビット・シフトがまったく必要ないこと，でき上がったデータを組み立てる処理もハードウェアで1クロックで行えること，などから，ソフトウェアよりも高速に処理できます．CPUの挙動を観察すると，APU命令のほうはデータ・アクセスを一気に行って4画素単位で処理しているのですが，CPU命令だけだと1画素データごとの処理を行っていることがわかります．

内部のキャッシュ・メモリは256ビット幅なので，8画素データ分をためておくことができます．すると，連続した2回目の4ワード・ロード命令はデータ・メモリをアクセスしていませんから，さらに高速にデータを送ることができます．

ただし，いくらAPU命令が有効であると言っても，この命令はユーザ回路との接続を行う手段の一つです．当然ながら，ユーザ回路のゲート規模分だけハードウェア・リソースを消耗します．今回の例では782スライスと三つのDSPブロックを消費しています．

また，この機能はソフトウェアなら30分程度で組めてしまうものなのですが，FCM_ABLEND回路の設計にはお

```
//    現時点ではAPU命令の後は転送量＋１回のNOPが必須である
#include “xbasic_types.h”
#include “xcache_l.h”
#include “xparameters.h”
#include “xpseudo_asm.h”
#include “xutil.h”

#include “APUinst.h”

// Data structures
volatile Xint32 __attribute__ ((aligned (32)))
                                        src2[640*480] ;
volatile Xint32 __attribute__ ((aligned (32)))
                                        dst2[640*480] ;

int main(void)
{
    Xint32 x,y ;
    Xint32 src_ptr, dst_ptr ;
    Xint32 srcPIX, dstPIX ;
    Xint32 srcA, srcR, srcG, srcB ;
    Xint32        dstR, dstG, dstB ;
    Xint32 resA, resR, resG, resB ;

  // initialize caches
    XCache_EnableICache(0x80000001);
                              // Instruction cache enable
    XCache_EnableDCache(0x80000001);
                              // Data cache enable

  // initialize APU
    mtmsr(XREG_MSR_APU_AVAILABLE);
                                  // コプロセッサ命令の有効化

    src_offset = 0x04000000 ;   // 転送元VRAMアドレスを指定
    dst_offset = 0x04800000 ;   // 転送先VRAMアドレスを指定

    for ( y=0; y<480; y++) {
        src_ptr = y*640 + src_offset ;
        dst_ptr = y*640 + dst_offset ;

        for ( x=0; x<32; x++) {

/*
            lwfcmx(2, src2, src_ptr);
 // CPU->UserHW : Load single x32 data. ( source PIXEL )
            _NOP_1 ;
            _NOP_1 ;     // 2回のNOPを明示的に指示
*/

/*
            ldfcmx(2, src2, src_ptr);
 // CPU->UserHW : Load double x32 data. ( source PIXEL )
            _NOP_1 ;
            _NOP_1 ;
            _NOP_1 ;     // 3回のNOPを明示的に指示
*/

            lqfcmx(2, src2, src_ptr);
   // CPU->UserHW : Load quad x32 data. ( source PIXEL )
            _NOP_1 ;
            _NOP_1 ;
            _NOP_1 ;
            _NOP_1 ;
            _NOP_1 ;     // 5回のNOPを明示的に指示

            src_ptr = src_ptr + 16 ;
                        // x32 x 4word pointer inclement.

            lqfcmx(3, dst2, dst_ptr);
                      //CPU->UserHW : Load quad x32 data.
                                   ( destination PIXEL )

            stqfcmx(4, dst2, dst_ptr);
                    // UserHW->CPU : Store quad x32 data.
                                   (distination PIXEL )

            dst_ptr = dst_ptr + 16 ;
                        // x32 x 4word pointer inclement.

        }

    }

  return 0;
}
```

（**b**）APU命令を使ったC言語記述

```
//    Non operation
#define     _NOP_1                 __asm__ __volatile__(\“nop “ \)
//    Load operation
#define lbfcmx(rn, base, adr)     __asm__ __volatile__(\“lbfcmx “ #rn “,%0,%1\n”\ : : “b” (base), “r” (adr)\)
#define lhfcmx(rn, base, adr)     __asm__ __volatile__(\“lhfcmx “ #rn “,%0,%1\n”\ : : “b” (base), “r” (adr)\)
#define lwfcmx(rn, base, adr)     __asm__ __volatile__(\“lwfcmx “ #rn “,%0,%1\n”\ : : “b” (base), “r” (adr)\)
#define ldfcmx(rn, base, adr)     __asm__ __volatile__(\“ldfcmx “ #rn “,%0,%1\n”\ : : “b” (base), “r” (adr)\)
#define lqfcmx(rn, base, adr)     __asm__ __volatile__(\“lqfcmx “ #rn “,%0,%1\n”\ : : “b” (base), “r” (adr)\)
//    Store operation
#define stbfcmx(rn, base, adr)     __asm__ __volatile__(\“stbfcmx “ #rn “,%0,%1\n”\ : : “b” (base), “r” (adr)\)
#define sthfcmx(rn, base, adr)     __asm__ __volatile__(\“sthfcmx “ #rn “,%0,%1\n”\ : : “b” (base), “r” (adr)\)
#define stwfcmx(rn, base, adr)     __asm__ __volatile__(\“stwfcmx “ #rn “,%0,%1\n”\ : : “b” (base), “r” (adr)\)
#define stdfcmx(rn, base, adr)     __asm__ __volatile__(\“stdfcmx “ #rn “,%0,%1\n”\ : : “b” (base), “r” (adr)\)
#define stqfcmx(rn, base, adr)     __asm__ __volatile__(\“stqfcmx “ #rn “,%0,%1\n”\ : : “b” (base), “r” (adr)\)
```

（**c**）APU命令を使うためのヘッダ

よそ3日ほどかかりました．多くの時間を要したのはFCBインターフェース回路の設計です．

● APU命令を使ってみて

今回いろいろとデータのサイズを変えるなどして評価しましたが，APU命令を適切に使わないと，逆にアプリケーションには合わないものも出てくると思いました．

APU命令はあくまでも，メモリ間のデータ入出力をドライバやOSの介在なしに行えるものです．例えば，

```
for(i = 0;i<4;i++) {
    *ptr++ = 0x12345678 ;
}
```

とすることはできません．0x12345678はあらかじめどこか

3

のメモリ領域(initial_value配列)に格納しておき,

```
lqfcmx( 0, *initial_value, 0 );
```

としてメモリ間転送にする必要があります.PowerPCのCPU命令で行うと,以下のようなコードになります.

```
for(i = 0;i<4;i++) {
  *ptr++ = *initial_value++ ;
}
```

ちなみにこのコードでは,1データを転送するだけで6クロックかかります(データ・フェッチのオーバヘッドは含まない).これはポインタのインデックス加算とデータ・セット,forによる範囲チェックが入るからであり,合計24クロックかかります.この結果,もしメモリ間転送を行うのであれば,1クロックで動作するlqfcmxを使ったほうが効率的であることがわかります.

APU命令が本領を発揮するのは,4ワード=128ビットごとに一気にデータの入出力を行うこと,アドレス・インデックス機能付き命令を使って,自動的にポインタを移動できること,データ・メモリ(データ・キャッシュ)とFCBインターフェースの間でデータ転送を行える場合であることを頭の片隅に入れておく必要があります.APU命令は,あるまとまったサイズのデータの塊をハードウェアに受け渡して結果を得るような場合には非常に有利に働きます.

4. まとめ

今回の例では,首尾よく7.56倍程度の加速性能を得ることができました.当初,筆者もこの程度は出るだろうと思っていたので,ひと安心です.半透明合成処理は,例題としては非常に簡単なものですが,データの飽和検出機能付きの乗算合成と半透明合成をソフトウェアで切り替えて使うなど,処理が重たくなるとこの差は開きます.10倍ぐらいの差が出るのではないかと予想しています.

注意するべき点は,これらの性能を得るためにはハードウェアもきちんと設計されている必要があり,ある程度まとまった量のデータを受け渡さないと,データの受け渡しだけで時間を消費してしまうことです.

ソフトウェアの性能を1として,ハードウェアの性能が100になるのなら,当然ハードウェアのほうがいいはずです.しかし,開発期間やコストも考えなければなりません.すべてのアプリケーションで100の性能を要求されるわけではありません.APU命令+ハードウェアというコプロセッサ機能は,性能やコストの最適点を得られる点でメリットがあるといえます.

APU命令の詳細はまだ非公開の部分も多く,使いこなすためにはあと半年程度はかかると思います.現在,筆者の会社をはじめとする数社で,FPGA-CPUワーキング・グループを立ち上げました.そこで,MPEG-2やJPEG2000を例に,APU命令の使用事例も含めた勉強会を開いており,知識データベースの蓄積を行っています.

参考・引用*文献

(1) *Accelerrated System Performance with the APU Controller and XtreamDSP Slice*, XAPP717, Xilinx, April 2005.
(2) *PowerPC Instruction Set Extension Guide*, ppc405_isaext_guide.pdf, Xilinx.
(3) *PowerPC405 APU controller*, *PowerPC405 Processor Block Reference Guide*, UG018.pdf, Xilinx.
(4) *PowerPC 405GP/GPr Embedded Processor*, AMPP.

いくら・まさみ
来栖川電工(有) 中野R&Dカンパニ
FPGAコンソーシアム(http://www.fpga.or.jp/)会員

<筆者プロフィール>
井倉将実.本稿執筆中にタイのタオ島で念願のダイビング免許を取得しました.機材の使いかたも専門用語もわからずヨチヨチ歩き状態ですが,Verilog HDL言語習得と同じように着実に身につけていこうと思っています.